### **Cahier des Charges d'Infrastructure, CI/CD et Opérations (CDC-DevOps)**

* **Version :** 1.0
* **Date :** 19 juin 2025
* **Statut :** En cours de rédaction
* **Propriétaire :** Ingénieur DevOps / Lead SRE

### **Partie 1 : Introduction et Philosophie DevOps**

#### **1.1. Objectif de ce Document**

Ce document est le **plan directeur et la référence unique** pour l'ensemble de l'infrastructure, des processus d'automatisation et des stratégies opérationnelles du projet SkillForge AI. Il a pour mission de traduire l'architecture logicielle en une plateforme technique fonctionnelle, en définissant de manière exhaustive comment nos services sont construits, testés, sécurisés, déployés et surveillés.

L'objectif ultime de ce CDC est de mettre en place un écosystème qui garantit les quatre piliers d'une plateforme moderne :

* **Fiabilité (Reliability) :** Assurer que la plateforme fonctionne comme prévu, avec un minimum de pannes.
* **Scalabilité (Scalability) :** Permettre à la plateforme de supporter une charge croissante d'utilisateurs et de données de manière transparente.
* **Sécurité (Security) :** Protéger la plateforme et ses données contre les menaces à chaque niveau.
* **Vélocité (Velocity) :** Donner aux équipes de développement la capacité de livrer de nouvelles fonctionnalités rapidement et en toute confiance.

#### **1.2. Audience Cible**

* **Audience Primaire :**
  + **Ingénieurs DevOps et SRE (Site Reliability Engineering) :** Ce document est leur manuel de construction et d'opération principal.
* **Audience Secondaire :**
  + **Développeurs (Back-End, Front-End, IA) :** Pour comprendre comment leur code est transformé en un service fonctionnel en production, et pour construire des applications compatibles avec nos standards opérationnels.
  + **Architectes Logiciels :** Pour valider que l'infrastructure et les processus de déploiement sont alignés avec la vision architecturale globale.
  + **Chefs de Projet Technique :** Pour comprendre les processus qui garantissent la stabilité de la plateforme et le rythme des livraisons.

#### **1.3. Principes Fondamentaux (Notre "Credo" DevOps)**

Notre approche DevOps repose sur quatre principes non-négociables qui dictent toutes nos décisions techniques et organisationnelles.

1. **Infrastructure en tant que Code (Infrastructure as Code - IaC)**
   * **Le Principe :** Toute ressource cloud, du réseau virtuel à la configuration d'un service, est définie de manière déclarative dans des fichiers de code (Terraform). La source de vérité de notre infrastructure est un dépôt Git, pas la console Google Cloud.
   * **La Conséquence :** Aucune modification manuelle de l'infrastructure ("click-ops") n'est autorisée sur les environnements de staging et de production. Cela garantit la reproductibilité, la traçabilité des changements et la capacité à reconstruire un environnement entier de manière automatisée.
2. **Automatisation Complète (CI/CD)**
   * **Le Principe :** Le chemin qui mène un commit de code à sa mise en production est 100% automatisé, prévisible et traçable. Chaque étape est exécutée par notre système d'Intégration et de Déploiement Continus (CI/CD).
   * **La Conséquence :** Les déploiements manuels sont considérés comme un échec de notre processus. L'automatisation réduit les erreurs humaines, accélère le cycle de livraison et permet aux développeurs de se concentrer sur la création de valeur.
3. **Sécurité Intégrée à Chaque Étape (DevSecOps)**
   * **Le Principe :** La sécurité n'est pas une étape finale ou un audit périodique, mais une propriété intégrée et automatisée tout au long du cycle de vie du logiciel ("Shift Left Security").
   * **La Conséquence :** Les scans de sécurité du code, des dépendances et des conteneurs sont des étapes obligatoires et bloquantes de nos pipelines. Une vulnérabilité critique détectée empêche le déploiement.
4. **Observabilité Totale (Total Observability)**
   * **Le Principe :** Chaque composant, chaque transaction, chaque action utilisateur doit générer des signaux observables : des **journaux (logs)**, des **métriques (metrics)** et des **traces**.
   * **La Conséquence :** Si un service n'est pas observable, il est considéré comme une boîte noire ingérable. L'observabilité est une exigence dès la phase de conception d'une fonctionnalité. Elle est la clé pour comprendre le comportement du système, déboguer rapidement les pannes et améliorer la performance.

Ces principes forment la culture et la discipline technique sur lesquelles nous allons bâtir la plateforme SkillForge AI.

### **Partie 2 : Gestion de l'Infrastructure (Infrastructure as Code)**

Cette section est l'implémentation directe de notre principe fondamental "Infrastructure en tant que Code" (IaC). Elle définit les outils et les stratégies que nous utilisons pour provisionner, configurer et gérer toutes les ressources sur notre fournisseur de cloud, Google Cloud Platform (GCP). L'objectif est de rendre notre infrastructure reproductible, versionnée et auditable.

#### **2.1. Outil IaC : Terraform**

1. **Technologie Obligatoire :** **Terraform** de HashiCorp est l'unique outil autorisé pour la gestion déclarative de notre infrastructure cloud.
2. **Justification :** Le choix de Terraform est motivé par plusieurs facteurs clés :
   * **Syntaxe Déclarative :** Nous décrivons l'état final souhaité de notre infrastructure, et Terraform se charge de calculer et d'appliquer les changements nécessaires pour y parvenir.
   * **Planification des Changements :** La commande terraform plan nous permet de visualiser et de valider toutes les modifications (créations, mises à jour, suppressions) avant de les appliquer, ce qui constitue un filet de sécurité essentiel.
   * **Écosystème Mature :** Terraform dispose d'un fournisseur Google Cloud (provider) robuste et d'une vaste communauté, garantissant la prise en charge de toutes les ressources GCP dont nous avons besoin.
   * **Modularité :** La capacité à créer des modules réutilisables nous permet de standardiser la création de nos composants (ex: un module pour créer un service Cloud Run avec sa configuration standard).

#### **2.2. Organisation et Gestion de l'État Terraform**

Le fichier d'état (.tfstate), qui contient la cartographie entre notre code et les ressources réelles, est un élément critique et sensible. Sa gestion doit être rigoureuse.

1. **Stockage de l'État (Backend) :**
   * **Règle Absolue :** Le fichier d'état ne doit **jamais** être stocké localement ou dans le dépôt Git.
   * **Implémentation :** Nous utiliserons un "backend" distant de type **Google Cloud Storage (GCS)**. Un bucket GCS dédié sera créé par Terraform pour stocker les fichiers d'état.
2. **Isolation des Environnements :**
   * **Règle :** Chaque environnement (dev, staging, prod) doit avoir son propre fichier d'état, complètement isolé des autres.
   * **Implémentation :** Nous utilisons une structure de répertoires qui crée un "workspace" implicite pour chaque environnement. Une modification appliquée sur l'environnement staging ne peut, par conception, avoir aucun impact sur l'environnement prod.
3. **Verrouillage de l'État (State Locking) :**
   * **Règle :** Pour éviter que deux ingénieurs appliquent des changements en même temps (ce qui corromprait l'état), le mécanisme de verrouillage doit être activé.
   * **Implémentation :** Le backend GCS de Terraform gère nativement le verrouillage d'état, garantissant qu'une seule opération apply peut être exécutée à un instant T sur un état donné.

#### **2.3. Catalogue des Ressources Gérées par Terraform**

La liste suivante est une énumération exhaustive des types de ressources GCP que nous gérerons via Terraform pour le MVP.

* **Organisation et Réseau :**
  + google\_project : Le projet GCP lui-même.
  + google\_project\_service : Pour l'activation des APIs nécessaires (Cloud Run API, GKE API, etc.).
  + google\_compute\_network : Notre réseau Virtual Private Cloud (VPC).
  + google\_compute\_subnetwork : Les sous-réseaux pour nos différentes régions.
  + google\_compute\_firewall : Les règles de pare-feu pour contrôler le trafic (ex: autoriser le trafic interne, bloquer l'accès externe aux bases de données).
* **Identités et Accès (IAM) :**
  + google\_service\_account : Un compte de service dédié pour chaque microservice et agent IA (ex: sa-project-service@..., sa-evaluation-agent@...).
  + google\_project\_iam\_member : Pour attribuer des rôles au niveau du projet.
  + google\_secret\_manager\_secret\_iam\_member : Pour autoriser un compte de service spécifique à accéder à un secret spécifique.
* **Services de Données :**
  + google\_sql\_database\_instance : L'instance PostgreSQL managée sur Cloud SQL.
  + google\_sql\_database : La base de données skillforge\_db au sein de l'instance.
  + google\_sql\_user : Les rôles de base de données pour chaque microservice (ex: project\_service\_role).
  + google\_redis\_instance : L'instance Memorystore pour notre broker de messages Redis.
* **Services Applicatifs et Orchestration :**
  + google\_cloud\_run\_v2\_service : La définition de chaque service applicatif (Back-End et Agents IA) sur Cloud Run.
  + google\_container\_cluster : Le cluster Google Kubernetes Engine (GKE) en mode Autopilot, destiné à l'orchestrateur de workflows Prefect.
* **Stockage et Registres :**
  + google\_storage\_bucket : Les buckets de stockage pour :
    - L'état de Terraform (tfstate).
    - Les fichiers statiques du Front-End.
    - Les fichiers téléversés par les utilisateurs (livrables, etc.).
    - Les modèles de Machine Learning.
    - Les jeux de données collectés pour l'entraînement.
  + google\_artifact\_registry\_repository : Le dépôt Docker privé pour nos images de conteneurs.
* **Messagerie et Secrets :**
  + google\_pubsub\_topic : La création des topics Pub/Sub nécessaires (ex: project.deliverable.submitted).
  + google\_secret\_manager\_secret : La création des "contenants" pour nos secrets (ex: mot de passe de la base de données, clé secrète JWT).

### **Partie 3 : Intégration et Déploiement Continus (CI/CD)**

Cette section est l'implémentation de notre principe "Automatisation Complète". Elle décrit la plateforme, la stratégie et les pipelines que nous utilisons pour intégrer, tester et déployer nos applications de manière rapide, fiable et sécurisée.

#### **3.1. Plateforme CI/CD : GitHub Actions**

1. **Technologie Obligatoire :** **GitHub Actions** est la seule et unique plateforme utilisée pour l'orchestration de nos pipelines CI/CD.
2. **Justification :** Ce choix est motivé par :
   * **Intégration Native :** Étant native à notre dépôt de code GitHub, la configuration des déclencheurs et l'intégration avec les Pull Requests sont transparentes.
   * **Configuration en tant que Code :** Les pipelines sont décrits dans des fichiers YAML (.github/workflows/\*.yml), versionnés aux côtés du code de l'application, ce qui les rend traçables et reproductibles.
   * **Écosystème Riche :** Le "Marketplace" de GitHub Actions offre des milliers d'actions réutilisables, nous permettant de composer des pipelines complexes sans réinventer la roue (ex: actions pour s'authentifier sur GCP, pour scanner des images Docker, etc.).

#### **3.2. Stratégie de Branches et Déclencheurs de Déploiement**

Notre flux de travail Git est conçu pour être simple et sécurisé, en associant des actions spécifiques à des branches clés.

* **Branches :**
  + **feature/\* :** Les branches de travail pour les nouvelles fonctionnalités et les corrections de bugs.
  + **develop :** La branche d'intégration principale. Elle représente la version la plus à jour de l'application et est la source de vérité pour l'environnement de **Staging**.
  + **main :** La branche stable. Elle ne contient que du code qui a été testé sur Staging et qui est considéré comme prêt pour la production. Elle est la source de vérité pour l'environnement de **Production**.
* **Déclencheurs (Triggers) :**
  + **Création d'une Pull Request vers develop :** Déclenche un pipeline de **validation** qui exécute les tests, les linters et les scans de sécurité. La fusion est techniquement bloquée si ce pipeline échoue.
  + **Merge sur la branche develop :** Déclenche le pipeline de **déploiement vers l'environnement de Staging**.
  + **Création d'un Tag de Release sur main :** Déclenche le pipeline de **déploiement vers l'environnement de Production**. Cette action est manuelle (créer un tag git comme v1.2.0) et garantit que les mises en production sont des événements délibérés et versionnés.

#### **3.3. Description Détaillée des Pipelines Types**

Voici le détail, étape par étape, des pipelines standards pour chaque type de composant de notre architecture.

##### **Pipeline pour un Microservice Back-End ou un Agent IA (Déploiement sur Cloud Run)**

1. **Déclenchement :** Push sur develop (Staging) ou Tag sur main (Production).
2. **Préparation :** Checkout du code et mise en place de l'environnement Python.
3. **Étape de Qualité (Quality Gate) :**
   * Exécution de la suite de tests (pytest).
   * Vérification du formatage et de la qualité du code (ruff, black).
   * Scan de sécurité des dépendances (pip-audit).
   * **Si l'une de ces étapes échoue, le pipeline s'arrête immédiatement.**
4. **Étape de Sécurité (Security Gate) :**
   * Analyse statique du code source à la recherche de vulnérabilités (SAST avec GitHub CodeQL).
5. **Construction de l'Artefact :**
   * Authentification à Google Artifact Registry.
   * Construction de l'image Docker via le Dockerfile du service.
   * Scan de l'image Docker construite pour détecter les vulnérabilités du système d'exploitation (ex: via Trivy ou le scanner natif de GCP).
6. **Publication de l'Artefact :** Push de l'image Docker validée vers Google Artifact Registry avec un tag unique (ex: le SHA du commit).
7. **Déploiement - Phase 1 (Base de Données) :**
   * Le pipeline s'authentifie à la base de données de l'environnement cible.
   * Il exécute la commande alembic upgrade head pour appliquer toutes les nouvelles migrations de schéma. **Si cette étape échoue, le déploiement est annulé.**
8. **Déploiement - Phase 2 (Application) :**
   * Le pipeline déploie une nouvelle révision du service sur Google Cloud Run en utilisant l'image Docker fraîchement publiée. Le trafic est basculé à 100% sur la nouvelle révision une fois qu'elle est déclarée saine.

##### **Pipeline pour l'Application Front-End (Déploiement sur GCS)**

1. **Déclenchement :** Push sur develop (Staging) ou Tag sur main (Production).
2. **Préparation :** Checkout du code et mise en place de l'environnement Node.js (npm ci).
3. **Étape de Qualité (Quality Gate) :**
   * Exécution de la suite de tests (vitest run).
   * Vérification de la qualité du code (eslint).
   * **Le pipeline échoue si une de ces étapes ne réussit pas.**
4. **Construction de l'Artefact :**
   * Exécution de la commande npm run build pour générer les fichiers statiques optimisés dans le répertoire /dist.
5. **Déploiement :**
   * Le pipeline s'authentifie à Google Cloud Storage.
   * Il synchronise le contenu du répertoire /dist avec le bucket GCS configuré pour l'hébergement web.
   * **Invalidation du Cache :** Il exécute une commande pour invalider le cache du Google Cloud CDN associé, afin que les utilisateurs reçoivent la nouvelle version immédiatement.

Ces pipelines d'automatisation constituent le cœur de notre usine logicielle. Ils nous permettent de livrer de la valeur de manière rapide et répétable, tout en intégrant la qualité et la sécurité à chaque étape.

### **Partie 4 : Sécurité de la Chaîne d'Approvisionnement (DevSecOps)**

Cette section est l'implémentation de notre principe "Sécurité Intégrée à Chaque Étape". Elle définit les outils et les processus automatisés que nous mettons en place pour sécuriser notre cycle de vie de développement logiciel, de l'écriture du code jusqu'au déploiement.

#### **4.1. Gestion Centralisée des Secrets**

La gestion des secrets (mots de passe, clés d'API, clés de signature JWT) est l'aspect le plus critique de notre sécurité opérationnelle.

1. **Règle Absolue : Zéro Secret dans le Code.**
   * Il est formellement interdit de stocker quelque secret que ce soit en clair dans : le code source, les fichiers de configuration, les variables d'environnement des pipelines CI/CD, ou les images Docker. Toute violation de cette règle est considérée comme une faille de sécurité critique.
2. **Outil Obligatoire : Google Secret Manager**
   * **Google Secret Manager** est l'unique source de vérité pour tous les secrets utilisés par nos applications dans les environnements de staging et production.
3. **Workflow de Gestion des Secrets :**
   * **Étape 1 - Provisionnement :** Les secrets sont créés et leurs valeurs sont ajoutées manuellement dans Google Secret Manager par un administrateur disposant des droits nécessaires.
   * **Étape 2 - Contrôle d'Accès (via Terraform) :** Le code Terraform ne contient jamais la valeur des secrets. Il contient uniquement les politiques IAM qui accordent au Service Account d'un service applicatif (ex: sa-project-service) le rôle Secret Manager Secret Accessor sur un secret spécifique.
   * **Étape 3 - Accès au Démarrage (Runtime) :** Au démarrage d'une instance de Cloud Run, l'application, s'exécutant avec son Service Account, utilise les librairies clientes de Google Cloud pour s'authentifier auprès de Secret Manager et récupérer en mémoire la valeur des secrets dont elle a besoin.

#### **4.2. Analyse de Sécurité des Images Docker**

Nos conteneurs sont basés sur des images de systèmes d'exploitation qui peuvent contenir des vulnérabilités connues (CVEs).

1. **Règle :** Chaque image Docker construite par notre pipeline CI/CD doit être scannée à la recherche de vulnérabilités connues avant d'être considérée comme éligible au déploiement.
2. **Outil :** Nous utilisons le **scanner de vulnérabilités natif de Google Artifact Registry**.
3. **Implémentation dans le Pipeline CI/CD :**
   * Après l'étape de push de l'image vers Artifact Registry, une étape du pipeline déclenche l'analyse.
   * Le pipeline est configuré pour **échouer et bloquer le déploiement** si le scan détecte des vulnérabilités de sévérité CRITICAL ou HIGH.

#### **4.3. Analyse Statique de la Sécurité du Code (SAST)**

Cette étape vise à détecter les failles de sécurité directement dans notre propre code applicatif.

1. **Règle :** L'intégralité du code source doit être analysée à la recherche de schémas de codage vulnérables à chaque Pull Request.
2. **Outil :** Nous utilisons **GitHub CodeQL**, qui est intégré à notre dépôt via GitHub Advanced Security.
3. **Implémentation dans le Pipeline CI/CD :**
   * Une action GitHub exécute l'analyse CodeQL pour chaque Pull Request ouverte contre la branche develop.
   * Les résultats sont affichés directement dans l'onglet "Security" de la Pull Request.
   * Les règles de protection de la branche develop sont configurées pour exiger le succès de ce scan avant d'autoriser la fusion.

#### **4.4. Analyse des Dépendances (Software Composition Analysis - SCA)**

Nos applications dépendent de nombreuses librairies open-source. Une faille dans l'une d'elles est une faille dans notre système.

1. **Règle :** Toutes les dépendances tierces (requirements.txt, package.json) doivent être continuellement surveillées et scannées à la recherche de vulnérabilités connues.
2. **Outils et Implémentation :** Nous utilisons une défense à deux niveaux :
   * **Analyse Continue (GitHub Dependabot) :** Dependabot est activé sur le dépôt. Il scanne en permanence nos dépendances et crée automatiquement des Pull Requests pour mettre à jour les paquets présentant des vulnérabilités connues.
   * **Analyse au Build (Pipeline CI/CD) :** Une étape de notre pipeline exécute une commande de scan explicite (pip-audit pour Python, npm audit pour le Front-End). Cette étape est configurée pour **échouer et bloquer le déploiement** si une vulnérabilité de sévérité haute ou critique est trouvée.

En intégrant ces quatre piliers de la sécurité directement dans nos pipelines automatisés, nous transformons la sécurité d'une corvée manuelle en une propriété intrinsèque et continue de notre processus de développement.

### **Partie 5 : Observabilité (Monitoring, Logging, Tracing)**

Cette section est l'implémentation de notre principe "Observabilité Totale". Elle définit la stratégie et les outils que nous utilisons pour collecter, visualiser et analyser les signaux émis par notre plateforme. Notre objectif est de passer d'une posture réactive (attendre que les utilisateurs signalent un problème) à une posture proactive (détecter et résoudre les problèmes avant qu'ils ne soient remarqués).

L'observabilité repose sur trois piliers : les Journaux (Logs), les Métriques (Metrics) et les Traces.

#### **5.1. Plateforme d'Observabilité**

1. **Technologie Standard :** La **suite opérationnelle de Google Cloud** (anciennement Stackdriver) est notre plateforme d'observabilité unique et centralisée.
2. **Justification :** Son intégration native avec l'ensemble de nos services GCP (Cloud Run, GKE, Cloud SQL, etc.) nous permet de collecter une grande partie des signaux sans configuration complexe. Elle fournit une interface unifiée pour les trois piliers de l'observabilité.

#### **5.2. Journalisation (Logging)**

* **Règle Absolue : Journalisation Structurée en JSON.**
  + Tous les services applicatifs (Back-End, Agents IA) et toutes les fonctions doivent émettre des logs au format JSON. Les logs en texte brut sont interdits car ils sont impossibles à analyser de manière programmatique et efficace.
* **Champs Obligatoires par Entrée de Log :**
  + Chaque log doit contenir au minimum les champs suivants pour permettre une corrélation et un filtrage efficaces dans Cloud Logging :
    - timestamp : La date et heure UTC exacte de l'événement.
    - severity : Le niveau de criticité (INFO, WARNING, ERROR, CRITICAL).
    - message : Le message lisible par un humain.
    - service\_name : Le nom du service émetteur (ex: project-service).
    - correlation\_id : L'identifiant de trace de la requête (voir section 5.4).

#### **5.3. Métriques et Monitoring**

* **Objectif :** Avoir une vision quantitative et en temps réel de la santé et de la performance de nos services.
* **Tableaux de Bord (Dashboards) :** Un tableau de bord dédié doit être créé dans **Cloud Monitoring** pour chaque environnement (staging, prod). Ce tableau de bord doit afficher au minimum les **"4 Golden Signals"** pour chaque service exposé à l'utilisateur :
  1. **Latence :** Le temps de réponse des requêtes, affiché aux 50ème, 95ème et 99ème percentiles.
  2. **Trafic :** Le nombre de requêtes par seconde (RPS).
  3. **Erreurs :** Le taux d'erreurs serveur (HTTP 5xx) en pourcentage du trafic total.
  4. **Saturation :** Le taux d'utilisation des ressources allouées (CPU, Mémoire) pour chaque service Cloud Run.
* **Métriques de la Base de Données :** Le tableau de bord de production doit également inclure les métriques clés de l'instance Cloud SQL : utilisation du CPU, de la mémoire, et nombre de connexions actives.

#### **5.4. Traçage Distribué (Tracing)**

* **Objectif :** Comprendre le cycle de vie complet d'une requête lorsqu'elle traverse plusieurs microservices et déclenche des événements asynchrones.
* **Implémentation :**
  + **Propagation du Contexte de Trace :** Le header HTTP **X-Cloud-Trace-Context**, automatiquement ajouté par le Load Balancer de Google, sera utilisé comme correlation\_id.
  + **Règle Applicative :** Chaque service doit :
    1. Extraire cet identifiant des requêtes entrantes.
    2. L'inclure dans tous ses logs structurés.
    3. Le propager dans les headers de tous les appels API sortants vers d'autres services internes.
    4. L'inclure dans le payload de tous les messages Pub/Sub qu'il publie.
* **Résultat :** Dans **Cloud Trace**, nous pourrons visualiser une cascade complète montrant le temps passé dans chaque service pour une seule requête utilisateur, nous permettant de diagnostiquer précisément les goulots d'étranglement.

#### **5.5. Stratégie d'Alertes (Alerting)**

* **Philosophie :** Nos alertes doivent être **actionnables** et **rares** pour éviter la "fatigue d'alerte". Nous privilégions les alertes basées sur les symptômes (impact utilisateur) plutôt que sur les causes.
* **Outil :** Toutes les politiques d'alerte sont configurées dans **Google Cloud Monitoring**.
* **Canaux de Notification :**
  + **#alerts-info (Slack) :** Pour les avertissements et les notifications de basse priorité.
  + **PagerDuty :** Pour les incidents critiques nécessitant une intervention humaine immédiate, 24/7.
* **Règles d'Alerte Initiales (pour la Production) :**
  + **Taux d'Erreurs :** SI le taux d'erreurs HTTP 5xx sur l'API Gateway dépasse 1% pendant 5 minutes, ALORS déclencher une alerte PagerDuty.
  + **Latence :** SI la latence au 99ème percentile dépasse 2000 ms pendant 10 minutes, ALORS envoyer une notification sur Slack.
  + **Utilisation de la Base de Données :** SI l'utilisation du CPU de l'instance Cloud SQL dépasse 90% pendant 15 minutes, ALORS déclencher une alerte PagerDuty.
  + **Facturation :** Une alerte de budget est configurée sur le projet GCP pour notifier les administrateurs si les dépenses prévisionnelles dépassent le seuil mensuel défini.

### **Conclusion du Document**

Ce Cahier des Charges DevOps établit la fondation sur laquelle repose l'ensemble de la plateforme SkillForge AI. En codifiant notre infrastructure, en automatisant nos déploiements, en intégrant la sécurité et en rendant le système entièrement observable, nous nous donnons les moyens de construire et d'opérer un produit de classe mondiale. L'adhésion à ces principes n'est pas une contrainte, mais le catalyseur de notre vélocité et de notre fiabilité.